Revised Report on the Algorithmic Language
ALGOL 60

PETER NAUR (Editor)

J. W. BACKUS  C. KATZ  H. RUTISHAUSER  J. H. WEGSTEIN
F. L. BAUER  J. McCARTHY  K. SAMELSON  A. VAN WINGAARDEN
J. GREEN  A. J. PERLIS  B. VAUQUOIS  M. WOODGER

Dedicated to the Memory of WILLIAM TURANSKI

SUMMARY

The report gives a complete defining description of the international algorithmic language ALGOL 60. This is a language suitable for expressing a large class of numerical processes in a form sufficiently concise for direct automatic translation into the language of programmed automatic computers.

The introduction contains an account of the preparatory work leading up to the final conference, where the language was defined. In addition, the notions, reference language, publication language and hardware representations are explained.

In the first chapter, a survey of the basic constituents and features of the language is given, and the formal notation, by which the syntactic structure is defined, is explained.

The second chapter lists all the basic symbols, and the syntactic units known as identifiers, numbers and strings are defined. Further, some important notions such as quantity and value are defined.

The third chapter explains the rules for forming expressions and the meaning of these expressions. Three different types of expressions exist: arithmetic, Boolean (logical) and designational.

The fourth chapter describes the operational units of the language, known as statements. The basic statements are: assignment statements (evaluation of a formula), go to statements (explicit break of the sequence of execution of statements), dummy statements, and procedure statements (call for execution of a closed process, defined by a procedure declaration). The formation of more complex structures, having statement character, is explained. These include: conditional statements, for statements, compound statements, and blocks.

In the fifth chapter, the units known as declarations, serving for defining permanent properties of the units entering into a process described in the language, are defined.

The report ends with two detailed examples of the use of the language and an alphabetic index of definitions.
INTRODUCTION

and the conference adopted this new form as the basis for its report. The Conference then proceeded to work for agreement on each item of the report. The present report represents the union of the Committee's concepts and the intersection of its agreements.

April 1962 Conference [Edited by M. Woodger]

A meeting of some of the authors of ALGOL 60 was held on April 2–3, 1962 in Rome, Italy, through the facilities and courtesy of the International Computation Centre. The following were present:

**Authors**

F. L. Bauer  
J. Green  
C. Katz  
R. Kogon

(representing J.W. Backus)

**Advisers**

M. Paul  
R. Franchetti  
P. Z. Ingerman

**Observer**

W. L. van der Poel  
(Chairman, IFIP TC 2.1 Working Group ALGOL)

P. Naur  
K. Samelson  
J. H. Wegstein  
A. van Wijngaarden  
M. Woodger

G. Seegmüller  
R. E. Utman  
P. Landin

The purpose of the meeting was to correct known errors in, attempt to eliminate apparent ambiguities in, and otherwise clarify the ALGOL 60 Report. Extensions to the language were not considered at the meeting. Various proposals for correction and clarification that were submitted by interested parties in response to the Questionnaire in ALGOL Bulletin No. 14 were used as a guide.

This report* constitutes a supplement to the ALGOL 60 Report which should resolve a number of difficulties therein. Not all of the questions raised concerning the original report could be resolved. Rather than risk hastily drawn conclusions on a number of subtle points, which might create new ambiguities, the committee decided to report only those points which they unanimously felt could be stated in clear and unambiguous fashion.

Questions concerned with the following areas are left for further consideration by Working Group 2.1 of IFIP, in the expectation that current work on advanced pro-

---

* [Editor's Note: The present edition follows the text which was approved by the Council of IFIP. Although it is not clear from the Introduction, the present version is the original report of the January 1960 conference modified according to the agreements reached during the April 1962 conference. Thus the report mentioned here is incorporated in the present version. The modifications touch the original report in the following sections: Changes of text: 1 with footnote; 2.1 footnote: 2.3; 2.7; 3.3.3; 3.3.4; 4.1.3; 4.2.3; 4.2.4; 4.3.4; 4.7.3; 4.7.3.1; 4.7.3.2; 4.7.5.1; 4.7.5.4; 4.7.6; 5; 5.3.3; 5.3.5; 5.4.3; 5.4.4; 5.4.5. Changes of syntax: 3.4.1; 4.1.1; 4.2.1; 4.5.1.]
programming languages will lead to better resolution:
1. Side effects of functions
2. The call by name concept
3. own: static or dynamic
4. For statement: static or dynamic
5. Conflict between specification and declaration

The authors of the ALGOL 60 Report present at the Rome Conference, being aware of the formation of a Working Group on ALGOL by IFIP, accepted that any collective responsibility which they might have with respect to the development, specification and refinement of the ALGOL language will from now on be transferred to that body.

This report has been reviewed by IFIP TC 2 on Programming Languages in August 1962 and has been approved by the Council of the International Federation for Information Processing.

As with the preliminary ALGOL report, three different levels of language are recognized, namely a Reference Language, a Publication Language and several Hardware Representations.

**Reference Language**

1. It is the working language of the committee.
2. It is the defining language.
3. The characters are determined by ease of mutual understanding and not by any computer limitations, coders notation, or pure mathematical notation.
4. It is the basic reference and guide for compiler builders.
5. It is the guide for all hardware representations.
6. It is the guide for transliterating from publication language to any locally appropriate hardware representations.

**Publication Language**

1. The main publications of the ALGOL language will use the reference representation.

**Hardware Representations**

1. Each one of these is a condensation of the reference language enforced by the limited number of characters on standard input equipment.
2. Each one of these uses the characters set of a particular computer and is the language accepted by a translator for that computer.
3. Each one of these must be accompanied by a special set of rules for transliterating from Publication or Reference language.

For transliteration between the reference language and a language suitable for publications, among others, the following rules are recommended.

<table>
<thead>
<tr>
<th>Reference Language</th>
<th>Publication Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscript bracket [ ]</td>
<td>Lowering of the line between the brackets and removal of the brackets</td>
</tr>
<tr>
<td>Exponentiation †</td>
<td>Raising of the exponent</td>
</tr>
<tr>
<td>Parentheses ( )</td>
<td>Any form of parentheses, brackets, braces</td>
</tr>
<tr>
<td>Basis of ten is</td>
<td>Raising of the ten and of the following integral number, inserting of the intended multiplication sign</td>
</tr>
</tbody>
</table>

**DESCRIPTION OF THE REFERENCE LANGUAGE**

1. Structure of the Language

As stated in the introduction, the algorithmic language has three different kinds of representations—reference, hardware, and publication—and the development described in the sequel is in terms of the reference representation. This means that all objects defined within the language are represented by a given set of symbols—and it is only in the choice of symbols that the other two representations may differ. Structure and content must be the same for all representations.

The purpose of the algorithmic language is to describe computational processes. The basic concept used for the description of calculating rules is the well-known arithmetic expression containing as constituents numbers, variables, and functions. From such expressions are compounded, by applying rules of arithmetic composition, self-contained units of the language—explicit formulae—called assignment statements.

To show the flow of computational processes, certain nonarithmetic statements and statement clauses are added which may describe, e.g., alternatives, or iterative repetitions of computing statements. Since it is necessary for the function of these statements that one statement refer to another, statements may be provided with labels. A sequence of statements may be enclosed between the statement brackets begin and end to form a compound statement.

Statements are supported by declarations which are not themselves computing instructions but inform the translator of the existence and certain properties of objects appearing in statements, such as the class of numbers taken on as values by a variable, the dimension of an

The reference language is built up from the following basic symbols:

\( (\text{basic symbol}) ::= (\text{letter})(\text{digit})(\text{logical value})(\text{delimiter}) \)

2.1. Letters

\( (\text{letter}) ::= a|b|c|d|e|f|g|h|i|j|k|i|m|n|o|p|q|r|s|t|u|v|w|x|y|z|A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z \)

This alphabet may arbitrarily be restricted, or extended with any other distinctive character (i.e., character not coinciding with any digit, logical value or delimiter).

Letters do not have individual meaning. They are used for forming identifiers and strings\(^6\) (cf. sections 2.4. Identifiers, 2.6. Strings).

2.2. Digits

\( (\text{digit}) ::= 0|1|2|3|4|5|6|7|8|9 \)

Digits are used for forming numbers, identifiers, and strings.

2.2.2. Logical Values

\( (\text{logical value}) ::= \text{true}/\text{false} \)

The logical values have a fixed obvious meaning.

2.3. Delimiters

\( (\text{delimiter}) ::= (\text{operator})(\text{separator})(\text{bracket})(\text{declarator})(\text{specifier}) \)

\( (\text{operator}) ::= (\text{arithmetic operator})(\text{relational operator})\)

\( (\text{arithmetic operator}) ::= +|-|\times|/|\div|\uparrow|\downarrow|\)\)

\( (\text{relational operator}) ::= <|\leq|>|\geq|\neq|\)

\( (\text{logical operator}) ::= \equiv|\tilde{\equiv}|\forall|\exists|\)\)

\( (\text{sequential operator}) ::= \text{go to}|\text{if}[\text{then}|\text{else}|\text{for}|\text{do})\)

\( (\text{separator}) ::= ;|,|\{\}|\mid\)\)

\( (\text{declarator}) ::= \text{step|until|while|comment}(\text{begin|end})(\text{procedure})(\text{specifier}) ::= \text{string|label|value} \)

Delimiters have a fixed meaning which for the most part is obvious or else will be given at the appropriate place in the sequel.

Typographical features such as blank space or change to a new line have no significance in the reference language. They may, however, be used freely for facilitating reading.

For the purpose of including text among the symbols of

\(^6\) It should be particularly noted that throughout the reference language underlining [in typewritten copy; boldface type in printed copy—Ed.] is used for defining independent basic symbols (see sections 2.2.2 and 2.3). These are understood to have no relation to the individual letters of which they are composed. Within the present report [not including headings—Ed.], boldface will be used for no other purpose.

\(^7\) do is used in for statements. It has no relation whatsoever to the do of the preliminary report, which is not included in ALGOL 60.
a program the following "comment" conventions hold:

The sequence of basic symbols: is equivalent to

begin comment (any sequence not containing ;); begin comment (any sequence not containing ;); begin comment (any sequence not containing end or ; or else) end

By equivalence is here meant that any of the three structures shown in the left-hand column may be replaced, in any occurrence outside of strings, by the symbol shown on the same line in the right-hand column without any effect on the action of the program. It is further understood that the comment structure encountered first in the text when reading from left to right has precedence in being replaced over later structures contained in the sequence.

2.4. IDENTIFIERS
2.4.1. Syntax

(identifier) ::= (letter)|(identifier)(letter)|(identifier)(digit)

2.4.2. Examples

q
Vita
a34tMNs
Marilyn

2.4.3. Semantics

Identifiers have no inherent meaning, but serve for the identification of simple variables, arrays, labels, switches, and procedures. They may be chosen freely (cf., however, section 3.2.4. Standard Functions).

The same identifier cannot be used to denote two different quantities except when these quantities have disjoint scopes as defined by the declarations of the program (cf. section 2.7. Quantities, Kinds and Scopes, and section 5. Declarations).

2.5. NUMBERS
2.5.1. Syntax

(unsigned integer) ::= (digit)|(unsigned integer)|(digit)

(integer) ::= (unsigned integer)+(unsigned integer)| (unsigned integer)

(decimal fraction) ::= .(unsigned integer)

(exponent part) ::= 10(integer)

(decimal number) ::= (unsigned integer)|(decimal fraction)| (unsigned integer)(decimal fraction)

(unsigned number) ::= (decimal number)|(exponent part)| (decimal number)(exponent part)

(number) ::= (unsigned number)+(unsigned number)| (unsigned number)

2.5.2. Examples

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>200.084</td>
<td>-0.083s-02</td>
<td></td>
</tr>
<tr>
<td>177</td>
<td>0.743 a 8</td>
<td>-17</td>
<td></td>
</tr>
<tr>
<td>5.894</td>
<td>9.34a+10</td>
<td>10-4</td>
<td></td>
</tr>
<tr>
<td>+0.7300</td>
<td>2-4</td>
<td>+u+5</td>
<td></td>
</tr>
</tbody>
</table>

2.5.3. Semantics

Decimal numbers have their conventional meaning. The exponent part is a scale factor expressed as an integral power of 10.

2.5.4. Types

Integers are of type integer. All other numbers are of type real (cf. section 5.1. Type Declarations).

2.6. STRINGS
2.6.1. Syntax

(proper string) ::= (any sequence of basic symbols not containing ' or ')|(empty)

(open string) ::= (proper string)|(open string)

(string) ::= (open string)

2.6.2. Examples

'skew" ["[ "| "=": "Tt"

"...This is a u u string"

2.6.3. Semantics

In order to enable the language to handle arbitrary sequences of basic symbols the string quotes ' and ' are introduced. The symbol u denotes a space. It has no significance outside strings.

Strings are used as actual parameters of procedures (cf. sections 3.2. Function Designators and 4.7. Procedure Statements).

2.7. QUANTITIES, KINDS AND SCOPES

The following kinds of quantities are distinguished: simple variables, arrays, labels, switches, and procedures.

The scope of a quantity is the set of statements and expressions in which the declaration of the identifier associated with that quantity is valid. For labels see section 4.1.3.

2.8. VALUES AND TYPES

A value is an ordered set of numbers (special case: a single number), an ordered set of logical values (special case: a single logical value), or a label.

Certain of the syntactic units are said to possess values. These values will in general change during the execution of the program. The values of expressions and their constituents are defined in section 3. The value of an array identifier is the ordered set of values of the corresponding array of subscripted variables (cf. section 3.1.4.1).

The various "types" (integer, real, Boolean) basically denote properties of values. The types associated with syntactic units refer to the values of these units.

3. Expressions

In the language the primary constituents of the programs describing algorithmic processes are arithmetic, Boolean, and designational expressions. Constituents of these expressions, except for certain delimiters, are logical values, numbers, variables, function designators, and elementary arithmetic, relational, logical, and sequential operators. Since the syntactic definition of both variables and function designators contains expressions, the definition of expressions, and their constituents, is necessarily recursive.

(expression) ::= (arithmetic expression)|(Boolean expression)|(designational expression)
3.1. VARIABLES

3.1.1. Syntax

(variable identifier) ::= (identifier)
(simple variable) ::= (variable identifier)
(subscript expression) ::= (arithmetic expression)
(subscript list) ::= (subscript expression)|(subscript list)
(subscript expression)
(array identifier) ::= (identifier)
(subscripted variable) ::= (array identifier)|(subscript list)
(variable) ::= (simple variable)|(subscripted variable)

3.1.2. Examples

epsilon
detA
a[7]
Q[7,2]
z[\sin(n \times pi/2),Q[3,4]]

3.1.3. Semantics

A variable is a designation given to a single value. This value may be used in expressions for forming other values and may be changed at will by means of assignment statements (section 4.2). The type of the value of a particular variable is defined in the declaration for the variable itself (cf. section 5.1, Type Declarations) or for the corresponding array identifier (cf. section 5.2, Array Declarations).

3.1.4. Subscripts

3.1.4.1. Subscripted variables designate values which are components of multidimensional arrays (cf. section 5.2, Array Declarations). Each arithmetic expression of the subscript list occupies one subscript position of the subscripted variable, and is called a subscript. The complete list of subscripts is enclosed in the subscript brackets [ ]. The array component referred to by a subscripted variable is specified by the actual numerical value of its subscripts (cf. section 3.3, Arithmetic Expressions).

3.1.4.2. Each subscript position acts like a variable of type integer and the evaluation of the subscript is understood to be equivalent to an assignment to this fictitious variable (cf. section 4.2.4). The value of the subscripted variable is defined only if the value of the subscript expression is within the subscript bounds of the array (cf. section 5.2, Array Declarations).

3.2. FUNCTION DESIGNATORS

3.2.1. Syntax

(procedure identifier) ::= (identifier)
(actual parameter) ::= (string)|(expression)|(array identifier)
(actual parameter) ::= (procedure identifier)
(letter string) ::= (letter)|(letter string)|(letter)
(parameter delimiter) ::= (string)
(actual parameter list) ::= (actual parameter)
(actual parameter list) ::= (actual parameter list)
(actual parameter)
(actual parameter part) ::= (empty)|(actual parameter list)
(function designator) ::= (procedure identifier)
(actual parameter part)

3.2.2. Examples

\sin(a-b)
J(n+s,n)
R
S(s-5)Temperature:(T):Pressure:(P)
Compile(\' := \':Stack:(Q)

3.2.3. Semantics

Function designators define single numerical or logical values, which result through the application of given sets of rules defined by a procedure declaration (cf. section 5.4, Procedure Declarations) to fixed sets of actual parameters. The rules governing specification of actual parameters are given in section 4.7, Procedure Statements. Not every procedure declaration defines the value of a function designator.

3.2.4. Standard functions

Certain identifiers should be reserved for the standard functions of analysis, which will be expressed as procedures. It is recommended that this reserved list should contain:

abs(E) for the modulus (absolute value) of the value of the expression E
sign(E) for the sign of the value of E (+1 for E>0, 0 for E=0, -1 for E<0)
sqrt(E) for the square root of the value of E
sin(E) for the sine of the value of E
cos(E) for the cosine of the value of E
cotan(E) for the principal value of the arctangent of the value of E
ln(E) for the natural logarithm of the value of E
exp(E) for the exponential function of the value of E (e^E).

These functions are all understood to operate indifferently on arguments both of type real and integer. They will all yield values of type real, except for sign(E) which will have values of type integer. In a particular representation these functions may be available without explicit declarations (cf. section 5, Declarations).

3.2.5. Transfer functions

It is understood that transfer functions between any pair of quantifies and expressions may be defined. Among the standard functions it is recommended that there be one, namely,

entier(E),

which "transfers" an expression of real type to one of integer type, and assigns to it the value which is the largest integer not greater than the value of E.

3.3. ARITHMETIC EXPRESSIONS

3.3.1. Syntax

(adding operator) ::= +|-
(multiplying operator) ::= \times|/|+|
(primary) ::= (unsigned number)|(variable)
(function designator)|(arithmetic expression)
(factor) ::= (primary)|(factor)|(primary)
(term) ::= (factor)|(term)|(multiplying operator)|(factor)
(simple arithmetic expression) ::= (term)
(adding operator)|(term)|(simple arithmetic expression)
(adding operator)|(term)
(if clause) ::= if(Boolean expression)then
(arithmetic expression) ::= (simple arithmetic expression)
(if clause)|(simple arithmetic expression)|else
(arithmetic expression)
3.3.2. Examples

Primitives:

\[
\begin{align*}
7.394_{10} & = 8 \\
\text{sum} & w[i+2,8]
\end{align*}
\]
\[
\begin{align*}
\cos(y+z×3) & \text{ sum } (a-3/y+vu/8)
\end{align*}
\]

Factors:

\[
\begin{align*}
\omega & = \cos(y+z×3)
\end{align*}
\]
\[
\begin{align*}
7.394_{10} & = 8 \text{ sum } [i+2,8] (a-3/y+vu/8)
\end{align*}
\]

Terms:

\[
\begin{align*}
U & = \omega \text{ sum } [\cos(y+z×3)/7.394_{10} = 8 \text{ sum } [i+2,8]]
\end{align*}
\]
\[
\begin{align*}
(a-3/y+vu/8)
\end{align*}
\]

Simple arithmetic expression:

\[
\begin{align*}
U - Y & = \omega \times \text{ sum } [\cos(y+z×3)/7.394_{10} = 8 \text{ sum } [i+2,8]]
\end{align*}
\]
\[
\begin{align*}
(a-3/y+vu/8)
\end{align*}
\]

Arithmetic expressions:

\[
\begin{align*}
w & = x-Q(S+Cw/2)
\end{align*}
\]
\[
\begin{align*}
if q > 0 & \text{ then } S+3xQ/A \text{ else } 2xS+3xq
\end{align*}
\]
\[
\begin{align*}
if a < 0 & \text{ then } U/V \text{ else if } a < 0 \text{ then } \text{ if } b > 1 \text{ then } U/V \text{ else if }
\end{align*}
\]
\[
\begin{align*}
k < y \text{ then } V/U \text{ else if } 0
\end{align*}
\]
\[
\begin{align*}
a & = \sin(\omega \times x)
\end{align*}
\]
\[
\begin{align*}
0.5712x4a(\times^{3}/(V-1)/2, 0)
\end{align*}
\]
\[
\begin{align*}
(4 \times \arctan(y+z/7+Q)
\end{align*}
\]
\[
\begin{align*}
if q & \text{ then } n-1 \text{ else } n
\end{align*}
\]
\[
\begin{align*}
if a < 0 \text{ then } A/B \text{ else if } b = 0 \text{ then } B/A \text{ else } z
\end{align*}
\]

3.3.3. Semantics

An arithmetic expression is a rule for computing a numerical value. In case of simple arithmetic expressions, this value is obtained by executing the indicated arithmetic operations on the actual numerical values of the primitives of the expression, as explained in detail in section 3.3.4 below. The actual numerical value of a primary is obvious in the case of numbers. For variables it is the current value (assigned last in the dynamic sense), and for function designators it is the value arising from the computing rules defining the procedure (cf. section 5.4.4. Values of Function Designators) when applied to the current values of the procedure parameters given in the expression. Finally, for arithmetic expressions enclosed in parentheses the value must through a recursive analysis be expressed in terms of the values of primitives of the other three kinds.

In the more general arithmetic expressions, which include if clauses, one out of several simple arithmetic expressions is selected on the basis of the actual values of the Boolean expressions (cf. section 3.4. Boolean Expressions). This selection is made as follows: The Boolean expressions of the if clauses are evaluated one by one in sequence from left to right until one having the value true is found. The value of the arithmetic expression is then the value of the first arithmetic expression following this Boolean (the largest arithmetic expression found in this position is understood). The construction:

else (simple arithmetic expression)

is equivalent to the construction:

else if true then (simple arithmetic expression)

3.3.4. Operators and Types

Apart from the Boolean expressions of if clauses, the constituents of simple arithmetic expressions must be of types real or integer (cf. section 5.1. Type Declarations). The meaning of the basic operators and the types of the expressions to which they lead are given by the following rules:

3.3.4.1. The operators +, -, and × have the conventional meaning (addition, subtraction, and multiplication). The type of the expression will be integer if both of the operands are of integer type, otherwise real.

3.3.4.2. The operations (term)/(factor) and (term) + (factor) both denote division, to be understood as a multiplication of the term by the reciprocal of the factor with due regard to the rules of precedence (cf. section 3.3.5). Thus for example

\[a/b \times 7/(p-q) \times r/s\]

means

\[((a\times(b^{-1})\times7)\times((p-q)^{-1})\times r)\times(a^{-1})\]

The operator / is defined for all four combinations of types real and integer and will yield results of real type in any case. The operator + is defined only for two operands both of type integer and will yield a result of type integer, mathematically defined as follows:

\[a+b = \text{sign}(a/b) \times \text{entier}(abs(a/b))\]

(cf. sections 3.2.4 and 3.2.5).

3.3.4.3. The operation (factor)^n(primary) denotes exponentiation, where the factor is the base and the primary is the exponent. Thus, for example,

\[21n16 \quad \text{means} \quad (2^1)^4\]

while

\[21(n1m) \quad \text{means} \quad 2^{n^m}\]

Writing i for a number of integer type, r for a number of real type, and a for a number of either integer or real type, the result is given by the following rules:

\[a^i\]

If i > 0, a×a×...×a (i times), of the same type as a.

If i = 0, a, of the same type as a.

If a = 0, undefined.

If i < 0, 1/(a×a×...×a) (the denominator has -i factors), of type real.

If a = 0, undefined.

\[a^r\]

If a > 0, exp(r×ln(a)), of type real.

If a = 0, if r > 0, 0.0, of type real.

If r ≤ 0, undefined.

If a < 0, always undefined.

3.3.5. Precedence of Operators

The sequence of operations within one expression is
generally from left to right, with the following additional rules:

3.3.5.1. According to the syntax given in section 3.3.1 the following rules of precedence hold:

- first: \( \uparrow \)
- second: \( \times \div + - \)
- third: \( \uparrow \)

3.3.5.2. The expression between a left parenthesis and the matching right parenthesis is evaluated by itself and this value is used in subsequent calculations. Consequently the desired order of execution of operations within an expression can always be arranged by appropriate positioning of parentheses.

3.3.6. Arithmetics of real quantities

Numbers and variables of type real must be interpreted in the sense of numerical analysis, i.e. as entities defined inherently with only a finite accuracy. Similarly, the possibility of the occurrence of a finite deviation from the mathematically defined result in any arithmetic expression is explicitly understood. No exact arithmetic will be specified, however, and it is indeed understood that different hardware representations may evaluate arithmetic expressions differently. The control of the possible consequences of such differences must be carried out by the methods of numerical analysis. This control must be considered a part of the process to be described, and will therefore be expressed in terms of the language itself.

3.4. Boolean Expressions

3.4.1. Syntax

- (relational operator) ::= \(<\mid\leq\mid=\mid\geq\mid\neq\)
- (relation) ::= (simple arithmetic expression)
- (function designator) ::= (relation)
- (Boolean primary) ::= (logical value) | (variable)
- (Boolean secondary) ::= (Boolean primary) | (Boolean primary)
- (Boolean factor) ::= (Boolean secondary) |
- (implication) ::= (Boolean factor) \(\rightarrow\) (Boolean term)
- (simple Boolean) ::= (implication)
- (Boolean expression) ::= (simple Boolean) else (Boolean expression)

3.4.2. Examples

- \( x = -2 \)
- \( Y > V \lor z < q \)
- \( a + b = -5 \land z - d > q \lor 2 \)
- \( p \land q \lor x = y \)
- \( g = a \land b \land c \lor d \lor e \lor f \)
- if \( t < 1 \) then \( a \) else \( k \leq c \)
- if \( t \) if \( a \) then \( b \) else \( c \) then \( d \) else \( f \) then \( g \) else \( h < k \)

3.4.3. Semantics

A Boolean expression is a rule for computing a logical value. The principles of evaluation are entirely analogous to those given for arithmetic expressions in section 3.3.3.

3.4.4. Types

Variables and function designators entered as Boolean primaries must be declared Boolean (cf. section 5.1. Type Declarations and section 5.4.4. Values of Function Designators).

3.4.5. The operators

Relations take on the value true whenever the corresponding relation is satisfied for the expressions involved, otherwise false.

The meaning of the logical operators \( \neg \) (not), \( \land \) (and), \( \lor \) (or), \( \Rightarrow \) (implies), and \( = \) (equivalent), is given by the following function table.

| \[ \neg \] | \[ b \] | \[ false \] | \[ false \] | \[ true \] | \[ true \] |
| \[ b \land b \] | \[ false \] | \[ false \] | \[ false \] | \[ false \] | \[ false \] |
| \[ b \lor b \] | \[ true \] | \[ true \] | \[ true \] | \[ true \] | \[ true \] |
| \[ b \Rightarrow b \] | \[ true \] | \[ true \] | \[ true \] | \[ true \] | \[ true \] |
| \[ b \Rightarrow b \] | \[ false \] | \[ false \] | \[ false \] | \[ false \] | \[ false \] |

3.4.6. Precedence of operators

The sequence of operations within one expression is generally from left to right, with the following additional rules:

3.4.6.1. According to the syntax given in section 3.4.1 the following rules of precedence hold:

- first: arithmetic expressions according to section 3.3.5.
- second: \( < \mid \leq \mid > \mid \geq \mid \neq \)
- third: \( \neg \)
- fourth: \( \land \)
- fifth: \( \lor \)
- sixth: \( \Rightarrow \)
- seventh: \( = \)

3.4.6.2. The use of parentheses will be interpreted in the sense given in section 3.3.5.2.

3.5. Designational Expressions

3.5.1. Syntax

- (label) ::= (identifier) (unsigned integer)
- (switch identifier) ::= (identifier)
- (switch designator) ::= (switch identifier) (subscript expression)
- (simple designational expression) ::= (label) (switch designator) (designational expression)
- (designational expression) ::= (simple designational expression) (if clause) (simple designational expression) else (designational expression)

3.5.2. Examples

- 17
- 0
- \( \text{Choose}[n-1] \)
- \( T \text{own}[: \text{if} y < 0 \text{ then} N \text{ else} N+1] \)
- \( \text{if} A < c \text{ then} 17 \text{ else} q \text{ if} u < 0 \text{ then} 2 \text{ else} n \)

3.5.3. Semantics

A designational expression is a rule for obtaining a label of a statement (cf. section 4. Statements). Again the principle of the evaluation is entirely analogous to that of arithmetic expressions (section 3.3.3). In the general case the Boolean expressions of the if clauses will select a simple designational expression. If this is a label the desired result is already found. A switch designator refers to the corresponding switch declaration (cf. section 3.3.
Switch Declarations) and by the actual numerical value of its subscript expression selects one of the designational expressions listed in the switch declaration by counting these from left to right. Since the designational expression thus selected may again be a switch designator this evaluation is obviously a recursive process.

3.5.4. The subscript expression

The evaluation of the subscript expression is analogous to that of subscripted variables (cf. section 3.1.4.2). The value of a switch designator is defined only if the subscript expression assumes one of the positive values 1, 2, 3, ..., n, where n is the number of entries in the switch list.

3.5.5. Unsigned integers as labels

Unsigned integers used as labels have the property that leading zeros do not affect their meaning, e.g. 00217 denotes the same label as 217.

4. Statements

The units of operation within the language are called statements. They will normally be executed consecutively as written. However, this sequence of operations may be broken by go to statements, which define their successor explicitly, and shortened by conditional statements, which may cause certain statements to be skipped.

In order to make it possible to define a specific dynamic succession, statements may be provided with labels.

Since sequences of statements may be grouped together into compound statements and blocks the definition of statement must necessarily be recursive. Also since declarations, described in section 5, enter fundamentally into the syntactic structure, the syntactic definition of statements must suppose declarations to be already defined.

4.1. Compound Statements and Blocks

4.1.1. Syntax

(unlabelled basic statement) ::= (assignment statement)

(go to statement) dummy statement) (procedure statement)

(basic statement) ::= (unlabelled basic statement) (label)

(basic statement) ::= (unconditional statement)

(compound statement) (block)

(statement) ::= (unconditional statement)

(conditional statement) (for statement)

(compound tail) ::= (statement) end (statement)

(compound tail)

(block head) ::= begin (declaration) (block head)

(declaration)

(unlabelled compound) ::= begin (compound tail)

(unlabelled block) ::= (block head) (compound tail)

(compound statement) ::= (unlabelled compound)

(label) (compound statement)

(block) ::= (unlabelled block) (label) (block)

(program) ::= (block) (compound statement)

This syntax may be illustrated as follows: Denoting arbitrary statements, declarations, and labels, by the letters S, D, and L, respectively, the basic syntactic units take the forms:

Compound statement:

L: L: ... begin S ; D ; ... S ; S ; ...S ; S end

Block:

L: L: ... begin D ; D ; ... D ; S ; S ; ...S ; S end

It should be kept in mind that each of the statements S may again be a complete compound statement or block.

4.1.2. Examples

Basic statements:

a := p+q

go to Naples

START: CONTINUE: W := 7.993

Compound statement:

begin x := 0 ; for y := 1 step 1 until n do

z := x + A[y];

if x > q then go to STOP else if x < 1 then
go to S ;

A[W] := x + bob end

Block:

Q: begin integer i, k ; real w ;

for i := 1 step 1 until m do

for k := 1 step 1 until n do

begin w := A[i, k];

A[i, k] := A[i, k];

A[i, k] := w end for i and k

end block Q

4.1.3. Semantics

Every block automatically introduces a new level of nomenclature. This is realized as follows: Any identifier occurring within the block may through a suitable declaration (cf. section 5. Declarations) be specified to be local to the block in question. This means (a) that the entity represented by this identifier inside the block has no existence outside it, and (b) that any entity represented by this identifier outside the block is completely inaccessible inside the block.

Identifiers (except those representing labels) occurring within a block and not being declared to this block will be nonlocal to it, i.e. will represent the same entity inside the block and in the level immediately outside it. A label separated by a colon from a statement, i.e. labelling that statement, behaves as though declared in the head of the smallest embracing block, i.e. the smallest block whose brackets begin and end enclose that statement. In this context a procedure body must be considered as if it were enclosed by begin and end and treated as a block.

Since a statement of a block may again itself be a block the concepts local and nonlocal to a block must be understood recursively. Thus an identifier, which is nonlocal to a block A, may or may not be nonlocal to the block B in which A is one statement.

4.2. Assignment Statements

4.2.1. Syntax

(left part) ::= (variable) := (procedure identifier) :=

(left part list) ::= (left part) (left part list) (left part)

(assignment statement) ::= (left part list) (arithmetic expression) :=

(left part list) (Boolean expression)
4.2.2. Examples

\[
s := p[0]; := n := n+1 + s
\]
\[
n := n+1
\]
\[
A := B/C - q \times S
\]
\[
S[0, k+2] := 3 - \arctan (s \times zeta)
\]
\[
V := Q > Y \wedge Z
\]

4.2.3. Semantics

Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may only occur within the body of a procedure defining the value of a function designator (cf. section 5.4.4). The process will in the general case be understood to take place in three steps as follows:

4.2.3.1. Any subscript expressions occurring in the left part variables are evaluated in sequence from left to right.
4.2.3.2. The expression of the statement is evaluated.
4.2.3.3. The value of the expression is assigned to all the left part variables, with any subscript expressions having values as evaluated in step 4.2.3.1.

4.2.4. Types

The type associated with all variables and procedure identifiers of a left part list must be the same. If this type is Boolean, the expression must likewise be Boolean. If the type is real or integer, the expression must be arithmetic. If the type of the arithmetic expression differs from that associated with the variables and procedure identifiers, appropriate transfer functions are understood to be automatically invoked. For transfer from real to integer type, the transfer function is understood to yield a result equivalent to

\[
\text{entier}(E+0.5)
\]

where E is the value of the expression. The type associated with a procedure identifier is given by the declarator which appears as the first symbol of the corresponding procedure declaration (cf. section 5.4.4).

4.3. Go To Statements

4.3.1. Syntax

(goto statement) ::= goto (designational expression)

4.3.2. Examples

\[
goto 8
\]
\[
goto exit [n+1]
\]
\[
goto Town [if y < 0 then N else N+1]
\]
\[
goto if A < C then 17 else 8 [if w < 0 then 2 else n]
\]

4.3.3. Semantics

A go to statement interrupts the normal sequence of operations, defined by the write-up of statements, by defining its successor explicitly by the value of a designational expression. Thus the next statement to be executed will be the one having this value as its label.

4.3.4. Restriction

Since labels are inherently local, no go to statement can lead from outside into a block. A go to statement may, however, lead from outside into a compound statement.

4.3.5. Go to an undefined switch designator

A go to statement is equivalent to a dummy statement if the designational expression is a switch designator whose value is undefined.

4.4. Dummy Statements

4.4.1. Syntax

(dummy statement) ::= (empty)

4.4.2. Examples

\[
L :\begin{align*}
\text{begin} & \quad : \quad \text{John} : \quad \text{end}
\end{align*}
\]

4.4.3. Semantics

A dummy statement executes no operation. It may serve to place a label.

4.5. Conditional Statements

4.5.1. Syntax

(if clause) ::= if (Boolean expression) then
(conditional statement) ::= (basic statement)
(compound statement) | (block)
(if statement) ::= if (if clause) (unconditional statement)
(if statement) ::= if (if clause) (conditional statement) (if clause) (else
statement) | (for statement) | (label) | (conditional statement)

4.5.2. Examples

if z > 0 then n := n + 1
if v > u then V := q := n + m else go to R
if s > 0 \checkmark P \leq Q then \forall x A : begin if q < v then a := v/s
else y := 2 \times a end
else if s > 0 then a := v - q else if v > s - 1
then go to S

4.5.3. Semantics

Conditional statements cause certain statements to be executed or skipped depending on the running values of specified Boolean expressions.

4.5.3.1. If statement. The unconditional statement of an if statement will be executed if the Boolean expression of the if clause is true. Otherwise it will be skipped and the operation will be continued with the next statement.

4.5.3.2. Conditional statement. According to the syntax two different forms of conditional statements are possible. These may be illustrated as follows:

if B1 then S1 else if B2 then S2 else S3 ; S4
and

if B1 then S1 else if B2 then S2 else if B3 then S3 ; S4

Here B1 to B3 are Boolean expressions, while S1 to S3 are unconditional statements. S4 is the statement following the complete conditional statement.

The execution of a conditional statement may be described as follows: The Boolean expression of the if clauses are evaluated one after the other in sequence from left to right until one yielding the value true is found. Then the unconditional statement following this Boolean is executed. Unless this statement defines its successor explicitly the next statement to be executed will be S4, i.e. the state-
ment following the complete conditional statement. Thus
the effect of the delimiter else may be described by saying
that it defines the successor of the statement it follows to
be the statement following the complete conditional
statement.

The construction

else (unconditional statement)

is equivalent to

else if true then (unconditional statement)

If none of the Boolean expressions of the if clause is true,
the effect of the whole conditional statement will be
equivalent to that of a dummy statement.

For further explanation the following picture may be
useful:

\[
\begin{array}{c}
\text{if } B_1 \text{ then } S_1 \text{ else if } B_2 \text{ then } S_2 \text{ else } S_3 \quad S_4 \\
B_1 \text{ false} & B_2 \text{ false}
\end{array}
\]

4.5.4. Go to into a conditional statement

The effect of a go to statement leading into a conditional
statement follows directly from the above explanation of
the effect of else.

4.6. For Statements

4.6.1. Syntax

(for list element) ::= (arithmetic expression)
(arithmetic expression) step (arithmetic expression) until
(Boolean expression)
(for list) ::= (for list element) (for list) (for list element)
(for clause) ::= for (variable) := (for list) do
(for statement) ::= (for clause) (statement)
(label) ::= (for statement)

4.6.2. Examples

for q := 1 step s until n do A[q] := B[q]
for k := 1, V1\times2 while V1 < N do
for j := I+G, L, 1 step L until N, C+D do
A[k,j] := B[k,j]

4.6.3. Semantics

A for clause causes the statement S which it precedes to
be repeatedly executed zero or more times. In addition it
performs a sequence of assignments to its controlled
variable. The process may be visualized by means of the
following picture:

\[
\begin{array}{c}
\text{Initialize} \quad \text{test} \quad \text{statement } S \\
\text{advance} \quad \text{successor} \quad \text{for list exhausted}
\end{array}
\]

In this picture the word initialize means: perform the first
assignment of the for clause. Advance means: perform the
next assignment of the for clause. Test determines if the
last assignment has been done. If so, the execution con-
tinues with the successor of the for statement. If not, the
statement following the for clause is executed.

4.6.4. The for list elements

The for list gives a rule for obtaining the values which
are consecutively assigned to the controlled variable. This
sequence of values is obtained from the for list elements
by taking these one by one in the order in which they are
written. The sequence of values generated by each of the
three species of for list elements and the corresponding
execution of the statement S are given by the following
rules:

4.6.4.1. Arithmetic expression. This element gives rise
to one value, namely the value of the given arithmetic
expression as calculated immediately before the corre-
sponding execution of the statement S.

4.6.4.2. Step-until-element. An element of the form
A step B until C, where A, B, and C, are arithmetic
expressions, gives rise to an execution which may be de-
scribed most concisely in terms of additional ALGOL
statements as follows:

\[
L_1: \text{if } (V-C) \times \text{sign}(B) > 0 \text{ then go to element exhausted} \\
\text{statement } S \\
\text{V := V+B} \\
goto L_1
\]

where V is the controlled variable of the for clause and
element exhausted points to the evaluation according to
the next element in the for list, or if the step-until-element
is the last of the list, to the next statement in the program.

4.6.4.3. While-element. The execution governed by a
for list element of the form E while F, where E is an
arithmetic and F a Boolean expression, is most concisely
described in terms of additional ALGOL statements as follows:

\[
L_3: \text{V := E} \\
\text{if } \neg F \text{ then go to element exhausted} \\
\text{statement } S \\
goto L_3
\]

where the notation is the same as in 4.6.4.2 above.

4.6.5. The value of the controlled variable upon exit

Upon exit out of the statement S (supposed to be com-
 pound) through a go to statement the value of the con-
trolled variable will be the same as it was immediately
preceding the execution of the go to statement.

If the exit is due to exhaustion of the for list, on the
other hand, the value of the controlled variable is unde-
defined after the exit.

4.6.6. Go to leading into a for statement

The effect of a go to statement, outside a for statement,
which refers to a label within the for statement, is unde-
defined.

4.7. Procedure Statements

4.7.1. Syntax

(actual parameter) ::= (string)/(expression)/(array identifier) |
switch identifier)/(procedure identifier)
(letter string) ::= (letter|letter string)(letter)
the procedure statement and the formal parameters of the procedure heading is established as follows: The actual parameter list of the procedure statement must have the same number of entries as the formal parameter list of the procedure declaration heading. The correspondence is obtained by taking the entries of these two lists in the same order.

4.7.5. Restrictions

For a procedure statement to be defined it is evidently necessary that the operations on the procedure body defined in sections 4.7.3.1 and 4.7.3.2 lead to a correct ALGOL statement.

This imposes the restriction on any procedure statement that the kind and type of each actual parameter be compatible with the kind and type of the corresponding formal parameter. Some important particular cases of this general rule are the following:

4.7.5.1. If a string is supplied as an actual parameter in a procedure statement or function designator, whose defining procedure body is an ALGOL 60 statement (as opposed to non-ALGOL code, cf. section 4.7.8), then this string can only be used within the procedure body as an actual parameter in further procedure calls. Ultimately it can only be used by a procedure body expressed in non-ALGOL code.

4.7.5.2. A formal parameter which occurs as a left part variable in an assignment statement within the procedure body and which is not called by value can only correspond to an actual parameter which is a variable (special case of expression).

4.7.5.3. A formal parameter which is used within the procedure body as an array identifier can only correspond to an actual parameter which is an array identifier of an array of the same dimensions. In addition if the formal parameter is called by value the local array created during the call will have the same subscript bounds as the actual array.

4.7.5.4. A formal parameter which is called by value cannot in general correspond to a switch identifier or a procedure identifier or a string, because these latter do not possess values (the exception is the procedure identifier of a procedure declaration which has an empty formal parameter part (cf. section 5.4.1) and which defines the value of a function designator (cf. section 5.4.4)). This procedure identifier is in itself a complete expression.

4.7.5.5. Any formal parameter may have restrictions on the type of the corresponding actual parameter associated with it (these restrictions may, or may not, be given through specifications in the procedure heading). In the procedure statement such restrictions must evidently be observed.

4.7.6. Deleted.

4.7.7. Parameter delimiters

All parameter delimiters are understood to be equivalent. No correspondence between the parameter delimiters used in a procedure statement and those used in the procedure heading is expected beyond their number being the
same. Thus the information conveyed by using the elaborated ones is entirely optional.

4.7.8. Procedure body expressed in code

The restrictions imposed on a procedure statement calling a procedure having its body expressed in non-
Algol code evidently can only be derived from the characteristics of the code used and the intent of the user and thus fall outside the scope of the reference language.

5. Declarations

Declarations serve to define certain properties of the quantities used in the program, and to associate them with identifiers. A declaration of an identifier is valid for one block. Outside this block the particular identifier may be used for other purposes (cf. section 4.1.3).

Dynamically this implies the following: at the time of an entry into a block (through the begin, since the labels inside are local and therefore inaccessible from outside) all identifiers declared for the block assume the significance implied by the nature of the declarations given. If these identifiers had already been defined by other declarations outside they are for the time being given a new significance. Identifiers which are not declared for the block, on the other hand, retain their old meaning.

At the time of an exit from a block (through end, or by a go to statement) all identifiers which are declared for the block lose their local significance.

A declaration may be marked with the additional declarator own. This has the following effect: upon a re-entry into the block, the values of own quantities will be unchanged from their values at the last exit, while the values of declared variables which are not marked as own are undefined. Apart from labels and formal parameters of procedure declarations and with the possible exception of those for standard functions (cf. sections 3.2.4 and 3.2.5), all identifiers of a program must be declared. No identifier may be declared more than once in any one block head.

Syntax.

(declaration) ::= (type declaration) | (array declaration) | (switch declaration) | (procedure declaration)

5.1. TYPE DECLARATIONS

5.1.1. Syntax

(type list) ::= (simple variable) | (type) | (type list)

(simple variable) ::= (local or own type) | (type) | (type declaration)

(type) ::= real | integer | Boolean

(local or own type) ::= (type) | own (type)

(type declaration) ::= (local or own type) | (type list)

5.1.2. Examples

integer p,q,r

own Boolean Acryl,n

5.1.3. Semantics

Type declarations serve to declare certain identifiers to represent simple variables of a given type. Real declared variables may only assume positive or negative values including zero. Integer declared variables may only assume positive and negative integral values including zero. Boolean declared variables may only assume the values true and false.

In arithmetic expressions any position which can be occupied by a real declared variable may be occupied by an integer declared variable.

For the semantics of own, see the fourth paragraph of section 5 above.

5.2. ARRAY DECLARATIONS

5.2.1. Syntax

(lower bound) ::= (arithmetic expression)

(upper bound) ::= (arithmetic expression)

(bound pair) ::= (lower bound) , (upper bound)

(bound pair list) ::= (bound pair) | (bound pair list), (bound pair)

(array segment) ::= (array identifier) , (bound pair list)

(array identifier), (array segment)

(array list) ::= (array segment) | (array list), (array segment)

(array declaration) ::= array (array list) , (local or own type)

array (array list)

5.2.2. Examples

array a, b, c[7:n,2:m], s[-2:10]

own integer array A [if c<0 then 2 else 1:20]

real array q[-7:-1]

5.2.3. Semantics

An array declaration declares one or several identifiers to represent multidimensional arrays of subscripted variables and gives the dimensions of the arrays, the bounds of the subscripts and the types of the variables.

5.2.3.1. Subscript bounds. The subscript bounds for any array are given in the first subscript bracket following the identifier of this array in the form of a bound pair list. Each item of this list gives the lower and upper bound of a subscript in the form of two arithmetic expressions separated by the delimiter ; The bound pair list gives the bounds of all subscripts taken in order from left to right.

5.2.3.2. Dimensions. The dimensions are given as the number of entries in the bound pair lists.

5.2.3.3. Types. All arrays declared in one declaration are of the same quoted type. If no type declarator is given the type real is understood.

5.2.4. Lower upper bound expressions

5.2.4.1 The expressions will be evaluated in the same way as subscript expressions (cf. section 3.1.4.2).

5.2.4.2. The expressions can only depend on variables and procedures which are nonlocal to the block for which the array declaration is valid. Consequently in the outermost block of a program only array declarations with constant bounds may be declared.

5.2.4.3. An array is defined only when the values of all upper subscript bounds are not smaller than those of the corresponding lower bounds.

5.2.4.4. The expressions will be evaluated once at each entrance into the block.

5.2.5. The identity of subscripted variables

The identity of a subscripted variable is not related to the subscript bounds given in the array declaration. How-
ever, even if an array is declared own the values of the corresponding subscripted variables will, at any time, be
defined only for those of these variables which have sub-
scripts within the most recently calculated subscript
bounds.

5.3. Switch Declarations

5.3.1. Syntax

\[(\text{switch list}) := (\text{designational expression}) |\
\hspace{1em} (\text{switch list}), (\text{designational expression}) |\
\hspace{1em} (\text{switch declaration}) := \text{switch} (\text{switch identifier}) := (\text{switch list})\]

5.3.2. Examples

\[
\text{switch } S := S_1, S_2, Q[m], \text{if } v > 5 \text{ then } S_3 \text{ else } S_4
\]
\[
\text{switch } Q := p_i, w
\]

5.3.3. Semantics

A switch declaration defines the set of values of the
corresponding switch designators. These values are given
one by one as the values of the designational expressions
entered in the switch list. With each of these designational
expressions there is associated a positive integer, 1, 2, ..., 
obtained by counting the items in the list from left to
right. The value of the switch designator corresponding to
a given value of the subscript expression (cf. section 3.5.
Designational Expressions) is the value of the designa-
tional expression in the switch list having this given value
as its associated integer.

5.3.4. Evaluation of expressions in the switch list

An expression in the switch list will be evaluated every
time the item of the list in which the expression occurs is
referred to, using the current values of all variables
involved.

5.3.5. Influence of scopes

If a switch designator occurs outside the scope of a
quantity entering into a designational expression in the
switch list, and an evaluation of this switch designator
selects this designational expression, then the conflicts
between the identifiers for the quantities in this expres-
sion and the identifiers whose declarations are valid at the
place of the switch designator will be avoided through
suitable systematic changes of the latter identifiers.

5.4. Procedure Declarations

5.4.1. Syntax

\[(\text{formal parameter}) := (\text{identifier}) |\
\hspace{1em} (\text{formal parameter list}) := (\text{formal parameter}) |\
\hspace{1em} (\text{formal parameter list}), (\text{parameter delimiter}) |\
\hspace{1em} (\text{formal parameter}) := (\text{empty}) | (\text{formal parameter list}) |\
\hspace{1em} (\text{identifier list}) := (\text{identifier}) | (\text{identifier list}), (\text{identifier}) |\
\hspace{1em} (\text{value part}) := (\text{value}) | (\text{identifier list}) |\
\hspace{1em} (\text{specifier}) := \text{string} | (\text{type}) | (\text{array}) | (\text{array}), (\text{identifier}) |\
\hspace{1em} (\text{procedure}) := \text{string} | (\text{type}), (\text{procedure}) |\
\hspace{1em} (\text{specifier part}) := (\text{empty}) | (\text{specifier}), (\text{identifier list}) |\
\hspace{1em} (\text{specification part}) := (\text{specifier}) | (\text{specification part}), (\text{identifier list}) |\
\hspace{1em} (\text{procedure heading}) := (\text{procedure}) |\
\hspace{1em} (\text{procedure body}) := (\text{statement}) | (\text{code}) |\
\hspace{1em} (\text{procedure declaration}) :=\
\hspace{1em} \text{procedure} (\text{procedure heading}), (\text{procedure body}) |\
\hspace{1em} (\text{type}) \text{ procedure} (\text{procedure heading}), (\text{procedure body})\]

5.4.2. Examples (see also the examples at the end of the report)

\[
\text{procedure } S\text{pur}(a) \text{Order:}(n) \text{Result:}(a) \text{ ; value } n \text{ ; array } a \text{ ; integer } n \text{ ; real } s \text{ ;} \\
\hspace{1em} \text{begin integer } k \text{ ;} \\
\hspace{1em} \text{s := 0 i ;} \\
\hspace{1em} \text{for } k := 1 \text{ step } 1 \text{ until } n \text{ do } s := s + a[k,k] \text{ end}
\]

\[
\text{procedure } \text{Transpose}(a) \text{Order:}(n) \text{ ; value } n \text{ ; array } a \text{ ; integer } n \text{ ;} \\
\hspace{1em} \text{begin real } w \text{ ; integer } i, k \text{ ;} \\
\hspace{1em} \text{for } i := 1 \text{ step } 1 \text{ until } n \text{ do} \\
\hspace{2em} \text{for } k := 1 + i \text{ step } 1 \text{ until } n \text{ do} \\
\hspace{3em} \text{begin w := a[i,k] ;} \\
\hspace{3em} a[i,k] := a[k,i] \\
\hspace{3em} a[k,i] := w \text{ end}
\]

\[
\text{end } \text{Transpose}
\]

\[
\text{integer procedure } \text{Step}(w) \text{ ; real } u \text{ ;} \\
\hspace{1em} \text{Step := if } 0 \leq u \wedge u \leq 1 \text{ then } i \text{ else 0}
\]

\[
\text{procedure } \text{Absmax}(a) \text{size:}(n,m) \text{Result:}(y) \text{Subscripts:}(i,k) \text{ ;} \\
\hspace{1em} \text{comment The absolute greatest element of the matrix } a \text{ ,} \\
\hspace{1em} \text{of size } n \text{ by } m \text{ transferred to } y \text{, and the subscripts of this} \\
\hspace{1em} \text{element to } i \text{ and } k \text{ .} \\
\hspace{1em} \text{array } a \text{ ; integer } n, m, i, k \text{ ; real } y \text{ ;} \\
\hspace{1em} \text{begin integer } p, q \text{ ;} \\
\hspace{1em} \text{y := 0 ;} \\
\hspace{1em} \text{for } p := 1 \text{ step } 1 \text{ until } n \text{ do} \\
\hspace{2em} \text{for } q := 1 \text{ step } 1 \text{ until } m \text{ do} \\
\hspace{3em} \text{if } \text{abs}(a[p,q]) > y \text{ then } \text{begin } y := \text{abs}(a[p,q]) \text{ ;} \\
\hspace{4em} i := p \text{ ;} \\
\hspace{4em} k := q \text{ end } \text{Absmax}
\]

\[
\text{procedure } \text{Innerproduct}(a,b) \text{Order:}(k,p) \text{Result:}(y) \text{ ; value } k \text{ ;} \\
\hspace{1em} \text{integer } k, p \text{ ; real } y, a, b \text{ ;} \\
\hspace{1em} \text{begin real } s \text{ ;} \\
\hspace{1em} \text{s := 0 ;} \\
\hspace{1em} \text{for } p := 1 \text{ step } 1 \text{ until } k \text{ do } s := s + a \times b \\
\hspace{1em} \text{y := s} \text{ end } \text{Innerproduct}
\]

5.4.3. Semantics

A procedure declaration serves to define the procedure
associated with a procedure identifier. The principal con-
stituent of a procedure declaration is a statement or a
piece of code, the procedure body, which through the use
of procedure statements and/or function designators may
be activated from other parts of the block in the head of
which the procedure declaration appears. Associated with
the body is a heading, which specifies certain identifiers
occurring within the body to represent formal parameters.
Formal parameters in the procedure body will, whenever
the procedure is activated (cf. section 3.2. Function
Designators and section 4.7. Procedure Statements) be
assigned the values of or replaced by actual parameters.
Identifiers in the procedure body which are not formal
will be either local or nonlocal to the body depending on
whether they are declared within the body or not. Those
of which them which are nonlocal to the body may well be
local to the block in the head of which the procedure decla-
ration appears. The procedure body always acts like a
block, whether it has the form of one or not. Consequently the scope of any label labelling a statement within the body or the body itself can never extend beyond the procedure body. In addition, if the identifier of a formal parameter is declared anew within the procedure body (including the case of its use as a label as in section 4.1.3), it is thereby given a local significance and actual parameters which correspond to it are inaccessible throughout the scope of this inner local quantity.

5.4.4. Values of function designators

For a procedure declaration to define the value of a function designator there must, within the procedure body, occur one or more explicit assignment statements with the procedure identifier in a left part; at least one of these must be executed, and the type associated with the procedure identifier must be declared through the appearance of a type declarator as the very first symbol of the procedure declaration. The last value so assigned is used to continue the evaluation of the expression in which the function designator occurs. Any occurrence of the procedure identifier within the body of the procedure other than in a left part in an assignment statement denotes activation of the procedure.

5.4.5. Specifications

In the heading a specification part, giving information about the kinds and types of the formal parameters by means of an obvious notation, may be included. In this part no formal parameter may occur more than once. Specifications of formal parameters called by value (cf. section 4.7.3.1) must be supplied and specifications of formal parameters called by name (cf. section 4.7.3.2) may be omitted.

5.4.6. Code as procedure body

It is understood that the procedure body may be expressed in non-Algol language. Since it is intended that the use of this feature should be entirely a question of hardware representation, no further rules concerning this code language can be given within the reference language.

Examples of Procedure Declarations:

**Example 1.**

```algol
procedure euler (fct, sum, eps, tim); value eps, tim;
integer tim; real procedure fct; real sum, eps;
comment euler computes the sum of fct(i) for i from zero up to infinity by means of a suitably refined euler transformation. The summation is stopped as soon as tim times in succession the absolute value of the terms of the transformed series are found to be less than eps. Hence, one should provide a function fct with one integer argument, an upper bound eps, and an integer tim. The output is the sum sum. euler is particularly efficient in the case of a slowly convergent or divergent alternating series;
begin integer i, k, n, t; array m[0:15]; real mn, mp, ds;
i := n := t := 0; m[0] := fct(0); sum := m[0]/2;
nextterm: i := i+1; mn := m[i];
for k := 0 step 1 until n do
  begin mp := (mn+mp)/2; m[k] := mn;
    mn := mp end;
end
```

**Example 2.**

```algol
procedure RK(y, n, FKT, eps, eta, z, E, B); value y, n;
integer n; Boolean B; real eps, eta, z, E;
procedure FKT;
comment: RK integrates the system y_i = f_i(x, y_1, y_2, ..., y_n) (k=1,2,...,n) of differential equations with the method of Runge-Kutta with automatic search for appropriate length of integration step. Parameters are: The initial values z and y[k] for x and the unknown functions y(x). The order n of the system. The procedure FKT(x,y,n,z) which represents the system to be integrated, i.e. the set of functions f_i. The tolerance values eps and eta which govern the accuracy of the numerical integration. The end of the integration interval x = E. The output parameter y[k] which represents the solution at z = E. The Boolean variable B, which must always be given the value true for an isolated or first entry into RK. If however the functions y must be available at several mesh-points z_0 z_1 z_2 ... z_n, then the procedure must be called repeatedly (with x = z_0, z = E = z_n, for k = 0, 1, ..., n-1) and then the later calls may occur with B = false which saves computing time. The input parameters of FKT must be x, y, n. The output parameter z represents the set of derivatives y[k] = f_i(x, y_1, y_2, ..., y[n]) for z and the actual y[k]. A procedure comp enters as a nonlocal identifier;
begin
  array z, y[1:3]; real z, y[1:3]; Boolean out;
  integer k, j; own real e, h;
  procedure RKST (z, x, y, z, E);
  real z, x, y; array e;
  comment: RKST integrates one single RUNGE-KUTTA with initial values z[k] which yields the output parameters z = z+h and y[k], the latter being the solution at z = E. Important: the parameters n, FKT, z enter RKST as nonlocal entities;
begin
  array w[1:n]; real a[1:5]; integer k, j;
x := z;
for k := 1 step 1 until n do y[k] := w[k] := y[k];
for j := 1 step 1 until 4 do
  begin
    FKT(z, w, y, z, E);
x := x + a[j];
    for k := 1 step 1 until n do
      begin
        w[k] := y[k] + a[j] * x[k];
y[k] := y[k] + a[j+1] * x[k]/3
      end
  end
end
```

* This RK-procedure contains some new ideas which are related to ideas of S. Grütz. A process for the step-by-step integration of differential equations in an automatic computing machine, [Proc. Camb. Phil. Soc. 47 (1951), 96]; and E. Falsenro, On the solution of ordinary differential equations with digital computing machines, [Fysikogr. Sällsk. Lund, Förh. 80, 11 (1950), 136-152]. It must be clear, however, that with respect to computing time and round-off errors it may not be optimal, nor has it actually been tested on a computer.

Communications of the ACM 15
REVIS Ed ALGOL 60

end k
end;
end RK1ST ;

Begin of program:
if f then begin H := xE-x ; s := 0 end else H := Hs ;
out := false ;
AA: if (x+2.0111H-xE>0) or (H>0) then begin Hs := H ; out := true ; H := (xE-x)/2
end if ;
RK1ST (x,y,2xH,2x1,y1) ;
BB: RK1ST (x,y,H,2x2,y2) ; RK1ST (x2y2H,2x3,y3) ;
for k := 1 step 1 until n do
if comp(y1[k],y0[k],eta) > eps then go to CC ;

comment: comp(a,b,c) is a function designator, the value of which is the absolute value of the difference of the mantissae of a and b. The exponents of these quantities have been made equal to the largest of the exponents of the originally given parameters a, b, c ;
x := x3 ; if out then go to DD ;
for k := 1 step 1 until n do y[k] := y3[k] ;
if s = 5 then begin s := 0 ; H := 2xH end if ;
s := s+1 ; go to AA ;
CC: H := 0.5xH ; out := false ; x1 := x2 ;
for k := 1 step 1 until n do y1[k] := y2[k] ;
go to BB ;
DD: for k := 1 step 1 until n do yE[k] := y3[k]
end RK

ALPHABETIC INDEX OF DEFINITIONS OF CONCEPTS AND SYNTACTIC UNITS

All references are given through section numbers. The references are given in three groups:

def Following the abbreviation "def", reference to the syntactic definition (if any) is given.
synt Following the abbreviation "synt", reference to the occurrences in metalinguistic formulae are given. References already quoted in the def-group are not repeated.
text Following the word "text", the references to definitions given in the text are given.

The basic symbols represented by signs other than underlined words [in typewritten copy; boldface in printed copy—Ed.] have been collected at the beginning.

The examples have been ignored in compiling the index.

+, see: plus
-, see: minus
×, see: multiply
/, ÷, see: divide
<, ≤, =, ≥, >, ≠, see: (relational operator)
=, ≡, ∨, ∧, ¬, see: (logical operator)
,, see: comma
., see: decimal point
\, see: ten
; , see: colon
; ;, see: semicolon
:=, see: colon equal
\, see: space
( ), see: parentheses
[ ], see: subscript brackets
', see: string quotes

(actual parameter), def 3.2.1, 4.7.1
(actual parameter list), def 3.2.1, 4.7.1
(actual parameter part), def 3.2.1, 4.7.1
(adding operator), def 3.3.1
alphabet, text 2.1
arithmetic, text 3.3.6
(arithmetic expression), def 3.3.1 synt 3, 3.1.1, 3.3.1, 3.4.1, 4.2.1,
\, 4.6.1, 5.2.1 text 3.3.3
(arithmetic operator), def 2.3 text 3.3.4
array, synt 2.3, 5.2.1, 5.4.1
array, text 3.1.4.1
(array declaration), def 5.2.1 synt 5 text 5.2.3
(array identifier), def 3.1.1 synt 3.2.1, 4.7.1, 5.2.1 text 2.8
(array list), def 5.2.1
(array segment), def 5.2.1
(array segment), def 4.2.1 synt 4.1.1 text 1, 4.2.3

(basic statement), def 4.1.1 synt 4.5.1
(basic symbol), def 2
begin, synt 2.3, 4.1.1
(block), def 4.1.1 synt 4.5.1 text 1, 4.1.3, 5
(block head), def 4.1.1
Boolean, synt 2.3, 5.1.1 text 5.1.3

Communication of the ACM

16
Supplement to the ALGOL 60 Report*

Edited by M. Woodger

Introduction

A meeting of some of the authors of ALGOL 60 was held on April 2-3, 1962 in Rome, Italy, through the facilities and courtesy of the International Computation Centre. The following were present:


Advisors: M. Paul, R. Franciotti, P. Z. Ingerman, G. Seegmuller, R. E. Utman, P. Landin

Observer: W. L. v. d. Poel (Chairman, IFIP TC 2.1, Working Group ALGOL)

The purpose of the meeting was to correct known errors in, attempt to eliminate apparent ambiguities in, and otherwise clarify the ALGOL 60 Report. Extensions to the language were not considered at the meeting. Various proposals for correction and clarification that were submitted by interested parties in response to the Questionnaire in ALGOL Bulletin No. 14† were used as a guide.

This report constitutes a supplement to the ALGOL 60 Report which should resolve a number of difficulties therein. Not all of the questions raised concerning the original report could be resolved. Rather than risk hastily drawn conclusions on a number of subtle points, which might create new ambiguities, the committee decided to report only those points which they unanimously felt could be stated in clear and unambiguous fashion.

Questions concerned with the following areas are left for further consideration by Working Group 2.1 of IFIP, in the expectation that current work on advanced programming languages will lead to better resolution: (1) side effects of functions, (2) the call-by-name concept, (3) own: static or dynamic, (4) for statement: static or dynamic, (5) conflict between specification and declaration.

The authors of the ALGOL 60 Report present at the Rome Conference, being aware of the formation of a Working Group on ALGOL by IFIP, accepted that any collective responsibility which they might have with respect to the development, specification and refinement of the ALGOL language will from now on be transferred to that body.

Misprints

The following corrections refer to misprints in the ALGOL 60 Report as published in the Communications of the ACM and Numerische Mathematik.

SECTION 3.3.4.3. [Comm. ACM and Numer. Math.] For read (the denominator has i factors)

the denominator has −i factors)

SECTION 3.4.1. [Comm. ACM] The formula for (implication) should read

(implication) ::= (Boolean term) \rightarrow (Boolean term)

SECTION 4.2.2. [Comm. ACM and Numer. Math.] The fourth example should begin with upper case S.

SECTION 4.7.1. [Comm. ACM] The formula for (parameter delimiter) should read

(parameter delimiter) ::= , | (letter string) :

Amendments

SECTION 1, paragraph 3, last sentence. Replace by

A sequence of statements may be enclosed between the statement brackets begin and end to form a compound statement.

SECTION 1, last two sentences of paragraph 4 and whole of paragraph 5. Replace by

A sequence of declarations followed by a sequence of statements enclosed between begin and end constitutes a block. Every declaration appears in a block in this way and is valid only for that block. A program is a block or compound statement which is not contained within another statement and which makes no use of other statements not contained within it.

SECTION 1, footnote 1. For is said to be undefined
read

is left undefined or said to be undefined.

SECTION 2.1, footnote [Comm. ACM edition]. Add
The reference language symbol for a space is \textit{u}. For typographical reasons the symbol \textit{s} is used here instead.

SECTION 2.3, last paragraph, following end. Replace by
By equivalence is here meant that any of the three structures shown in the left hand column may, in any occurrence outside of strings, be replaced by the symbol shown on the same line in the right hand column without any effect on the action of the program. It is further understood that the comment structure encountered first in the text when reading from left to right has precedence in being replaced over later structures contained in the sequence.

SECTION 2.7., second paragraph. Replace by
The scope of a quantity is the set of statements and expressions in which the declaration of the identifier associated with that quantity is valid. For labels see section 4.1.3.

SECTION 3.3.3. For
(cf. section 5.4 Procedure declarations)

read
(cf. section 5.4.4. Values of function designators)

SECTION 3.3.4.2. For
defined as follows

read

mathematically defined as follows.

SECTION 3.4.1. Replace the formula for \textit{relation} by
\begin{align*}
\text{(relation)} &::= \text{(simple arithmetic expression)} \\
&\text{(relational operator)} \text{(simple arithmetic expression)}
\end{align*}

SECTION 4.1.1. Add after the formula for \textit{block} the following:
\begin{align*}
\text{(program)} &::= \text{(block)} \mid \text{(compound statement)}
\end{align*}

SECTIONS 4.1.1 and 4.5.1. Replace the formulae for \textit{(unconditional statement)} and \textit{(statement)} respectively by the following:
\begin{align*}
\text{(unconditional statement)} &::= \text{(basic statement)} \mid \text{(compound statement)} \mid \text{(block)} \\
\text{(statement)} &::= \text{(unconditional statement)} \mid \text{(conditional statement)} \mid \text{(for statement)}
\end{align*}

SECTION 4.1.3., paragraph 2, last sentence. Replace by:
A label separated by a colon from a statement, i.e. labelling that statement behaves as though declared in the head of the smallest enclosing block, i.e. the smallest block whose brackets begin and end enclose that statement. In this context a procedure body must be considered as if it were enclosed by begin and end and treated as a block.

SECTION 4.2.1. Replace the formula for \textit{(left part)} by:
\begin{align*}
\text{(left part)} &::= \text{(variable)} \mid \text{(procedure identifier)}
\end{align*}

SECTION 4.2.3. Replace the first sentence by
Assignment statements serve for assigning the value of an expression to one or several variables or procedure identifiers. Assignment to a procedure identifier may only occur within the body of a procedure defining the value of a function designator (cf. section 5.4.4.).

SECTION 4.2.4., first four statements. Replace by
The type associated with all variables and procedure identifiers of a left part list must be the same. If this type is Boolean, the expression must likewise be Boolean. If the type is real or integer, the expression must be arithmetic. If the type of the arithmetic expression differs from that associated with the variables and procedure identifiers, appropriate transfer functions are understood to be automatically invoked.

SECTION 4.2.4. Add:
The type associated with a procedure identifier is given by the declarator which appears as the first symbol of the corresponding procedure declaration (cf. section 5.4.4.).

SECTION 4.3.4. Add:
A \textit{go to statement} may, however, lead from outside into a compound statement.

SECTION 4.5.1. Replace the formula for \textit{(if statement)} by:
\begin{align*}
\text{(if statement)} &::= \text{(if clause)} \text{(unconditional statement)}
\end{align*}
and the formula for \textit{(conditional statement)} by
\begin{align*}
\text{(conditional statement)} &::= \text{(if statement)} \mid \text{(else statement)} \mid \text{(if clause)} \text{(for statement)} \mid \text{(label)} \mid \text{(conditional statement)}
\end{align*}

SECTION 4.7.3. Replace the final colon by
at the time of execution of the procedure statement:

SECTION 4.7.3.1. Replace the last sentence by
The effect is as though an additional block embracing the procedure body were created in which those assignments were made to variables local to this fictitious block with types as given in the corresponding specifications (cf. section 5.4.5.). As a consequence, variables called by value are to be considered as nonlocal to the body of the procedure, but local to the fictitious block (cf. section 5.4.3.).

SECTION 4.7.3.3. Add:
If the procedure is called from a place outside the scope of any nonlocal quantity of the procedure body the conflicts between the identifiers inserted through this process of body replacement and the identifiers whose declarations are valid at the place of the procedure statement or function designator will be avoided through suitable systematic changes of the latter identifiers.

SECTION 4.7.5.1. Replace by:
If a string is supplied as an actual parameter in a procedure statement or function designator, whose defining procedure body is an \texttt{ALGOL 60} statement (as opposed to non-\texttt{ALGOL} code, cf. section 4.7.8), then this string can only be used within the procedure body as an actual parameter in further procedure calls. Ultimately it can only be used by a procedure body expressed in non-\texttt{ALGOL} code.

SECTION 4.7.5.4. For

\text{a switch identifier or a procedure identifier,}
Suggestions On
ALGOL 60 (ROME) Issues

A Report by the American Standards Association Subcommittee X3.4.2

[Editorial Note: The following report by the USA ASA X3.4.2 Working Group on Programming Language Specifications contains suggested specific solutions to the unresolved issues of the “Supplement to the ALGOL 60 Report” [Comm. ACM 6 (Jan. 1963), 18-20; (this issue)]. This report is published in this section in order to give these proposals more comprehensive USA consideration and stimulation of support or counter proposals. This is to be considered as a recommendation to the X3.4 subcommittee and not as the official position of the X3.4 subcommittee—Julien Green]

X3.4.2/1, 30 September 1962

SUBJECT: Suggestions on ALGOL 60 (ROME) Issues

At the IFIP WG 2.1 meeting in Munich it was recommended that the ALGOL report as modified by the Rome authors’ meeting be submitted as an ISO standard programming language. At the same time it was recommended that a subset be defined which would form a part of the standard.

Since there were five problem areas defined, but not solved, at Rome, X3.4.2 felt that suggestions on their resolution should be prepared for each of these. Moreover, X3.4.2 feels that though ALGOL is satisfactory as a publication language, it would be absolutely necessary to develop an input-output facility for the language before it should be considered as a standard program language.

This document contains a brief discussion of the five problem areas and other issues, together with recommended positions on each. We recommend that the problems of input-output be returned to IFIP WG2.1 for solution.

Finally, it is an assumption in X3.4.2 (which we wish made explicit) that implementation of a standard ALGOL subset would in no way imply implementation of full ALGOL 60 (ROME).

The following issues and problem areas of ALGOL 60 (ROME) are treated herein:
1. Side effects in function designators.
2. General versus restricted “call by name”
3. “Own” variables and arrays
4. Static versus dynamic “for” statements
5. Conflict between Specification and Declaration

Also considered are:

a. Function designator call statements
b. Abnormal exits from function designators
c. Numeric labels
d. Initialization of “own” variables

(Signed) R. E. UTMAN
Chairman, X3.4.2

1. Side Effects in Function Designators

The term “side effects” is used to designate the ability of procedures and function designators to change the values of variables which are nonlocal to themselves. This ability, when properly controlled, can add a great deal of power to the language; and few question it in connection with procedure-call statements.
When side effects are allowed for function designators, however, a number of undesirable consequences occur:

1. The strict order of evaluation of primaries within statements must be defined.

2. It is necessary to evaluate all function designators in Boolean expressions even though the truth value may be known after the evaluation of the first term.

It would appear then, desirable to eliminate side effects from function designators. Unfortunately, though, it does not solve the problems mentioned in 1 and 2. There is another construct, that of own, which causes the same misbehavior. An own variable essentially acts as a memory element for a function or procedure, and hence the order of evaluation must still be specified and all function designators still evaluated so that the appropriate "history" will be contained in these own variables.

Rather than eliminating this concept from the language, we would recommend that:

The order of evaluation of primaries within a statement be left to rights as written, and all function designators in a statement which can cause side effects or the modification of own variables must be evaluated.

Side effects in function designators, and own variable should be eliminated from the official Algol subset.

2. General Versus Restricted CALL by Name

The only argument that has been advanced against the general call by name can be reduced to one of inefficiency in implementation. Since it is not felt that this is a valid consideration (and, indeed, was specifically eliminated as a point of discussion during the writing of the Algol 60 report) it seems that the only logical alternative is to endorse the general call by name.

That this was present in the minds of at least some of the Algol 60 authors is evidenced by the procedure Innerproduct, given in 5.4.2 of the Algol 60 report.

A technique for implementing this general call by name has been given in the literature [Ingerman, P. Z. "Thunks." Comm. ACM 4 (Jan. 1961)]; hence the argument that it is unimplementable is vacuous.

3. Own Variables and Arrays

3a. Own variables and arrays in Algol are essentially a memory device. They are a means whereby a "block" can retain a "measure" of its past history. The mechanism is that any variable marked own on re-entry of a block, will retain the value it had at last exit. This is as opposed to local variables not marked own, whose values are undefined at each entry.

Unfortunately there are two major difficulties with respect to own variables: (1) Algol contains no initialization facilities for them; (2) Their exact behavior is ambiguous.

The first of these objections could be handled quite easily by adding some initialization facility to the language. A possible way would be analogous to the switch declarator:

```
eg. own integer X:=3, Y:=5, Z:=M;
```

The second of these points will require considerable discussion.

Two major interpretations of own have arisen—the dynamic interpretation, and the static interpretation. The dynamic interpretation associates the own declaration with a block as executed, the static interpretation, with a block as written. These two interpretations are in agreement for the case that the block is not contained in a procedure. When the block does form part of a procedure, however, the results of the two interpretations are widely divergent.

We will discuss the implications of these two interpretations in light of the following example:

```
begin procedure P;
    begin own real X;
        LA: P;
        LB: P;
        end;
        LC: P;
        LD: P;
        end;
```

3b. Dynamic Interpretation. The dynamic interpretation is based primarily on the description of the behavior of a procedure statement (4.7.3)—"the procedure body . . . is inserted in place of the procedure statement and executed." Because of recursive procedures, and procedures as parameters, this replacement cannot be done during compilation. However, we will consider this replacement to occur just prior to the time of execution of the procedure statement.

In the example there are two places in the main program where the procedure P is called—LC and LD. At each of these places we insert the procedure body in place of the procedure statement. We now have two new blocks in the program labeled LC and LD, each of which has an own variable X. Let us differentiate them according to the block labels as $X_{c}$ and $X_{d}$. According to the dynamic interpretation there are two distinct variables, each of which must be remembered from execution to execution of the blocks, and which can have no effect on each other even though they were formed by calls in the same procedure. Now the bodies of the procedure P contain two calls on P at LA and LB. If we continue the process which we described above we obtain two new own X's which we can designate $X_{ac}$ and $X_{ad}$ belonging to the block labeled LC and two others in LD—$X_{bc}$ and $X_{bd}$. After three levels of recursion we have the following own variables defined, all of which are distinct and which must be retained for the next entry to their block.

```
A similar structure would be built up for LD.
```
3c. **Static Interpretation.** Since the static interpretation associates the own variable with the typographical block, there would be exactly one X defined which would be referenced by all calls on the procedure. In effect, the own variable behaves as though it were nonlocal to the entire program, and the own declaration makes it available to a block. (Note that this is exactly why the own concept causes the same misbehavior as the changing of nonlocal quantities, i.e., side effects.)

3d. **Static or Dynamic?** At first glance it would appear that the dynamic interpretation of own is the “better” on the basis of “generality” (the touchstone of language designers and the bane of implementors). However, it is sometimes very easy to confuse complexity with generality, and I think this is the case here. We said earlier that the purpose of the own variable was to record the history of a block. Under the dynamic interpretation we are not recording the history of the procedure, but the history of a particular call. Thus, if we had a procedure RANDOM which generated random numbers, keeping track of its history through the use of own variables, we would obtain the answer 0 for RANDOM—RANDOM under the dynamic interpretation, rather than a new random number which we would expect from the definition of random numbers—and would indeed obtain from the static interpretation. It is just the independency of the variable under the dynamic interpretation which destroys its utility.

Therefore we recommend that:

The static interpretation of own variables be the one accepted for an Algol standard and that the description of the own concept be rewritten to indicate clearly its nonlocal behavior and to abrogate any possible conflict with the description of procedure statement.

4. **Static Versus Dynamic for Statements**

An ambiguity exists in the interpretation of a for statement whose (for list) elements contain expressions which may change their values during the execution of the for statement.

Although this ambiguity exists conjecturally (excluding side effects) but not actually for those elements which are arithmetic expressions, a very real ambiguity arises for both the increment and the upper limit in the step-until-element and for the limit in the while-element.

Consider, therefore, the step-until-element “A step B until C”. The arithmetic expression “A” will be evaluated precisely once, when the loop is initialized; hence, as when the for list element is an arithmetic expression, it cannot effectively change during the execution of the for statement. This leaves four alternatives:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>a.</td>
<td>static static</td>
</tr>
<tr>
<td>b.</td>
<td>static dynamic</td>
</tr>
<tr>
<td>c.</td>
<td>dynamic static</td>
</tr>
<tr>
<td>d.</td>
<td>dynamic dynamic</td>
</tr>
</tbody>
</table>

By comparing “B” with the arithmetic expression of a while-element, it is apparent that considering that “B” be evaluated dynamically gives the closest analogy. (Note that the while-element would be of minimal utility if the arithmetic expression were to be evaluated precisely once, and that value used thereafter.) This analogy eliminates case a and case b from the above listing.

Decision between case c and case d is somewhat more arbitrary, but it is felt that it is not in the spirit of the original report to restrict unnecessarily; hence case d is recommended as the position to be taken.

It is also suggested that the interpretation of the step-until-element might be made more efficient by replacing the Algol 60 statements in 4.6.4.2 with:

```plaintext
S1 := V := A;
S2 := B;
L1: if S2 × (S1 − C) > 0 then go to element exhausted;
   Statement S;
S2 := B;
S1 := V := V + S2;
go to L1;
```

5. **Conflict Between Specification and Declaration**

A conflict between the specification and declaration of a variable arises when a variable has been identified (by declaration) as being of one type; is used as an actual parameter in a procedure statement; and the corresponding formal parameter is specified in the procedure declaration to be of some other type.

There are three cases in which conflict arises. In case I the conflict is between incompatible types (e.g. a Boolean declaration and a label specification). Case II arises when no declaration appears, but only a specification. Case III involves compatible types where both declaration and specification appear.

When the types are incompatible the most natural resolution of the conflict is to declare that the situation is undefined; in fact, if the category of “programming error” were to exist in Algol 60, this conflict would be an example. The table below gives a list of specifiers; the list of declarators is a subset. Each line of the table is asserted to be incompatible (in the sense of this paragraph) with all other lines, but not with other entries on the same line.

- string
- real integer
- Boolean
- array real array integer array
- Boolean array
- label
- switch
- procedure
- real procedure integer procedure
- Boolean procedure

Hence, the recommended position for case I is that the program is undefined if the declaration and specification appear on different lines of the above table.

Case II can be reduced to either case I or case III by noting that only labels, numbers and expressions do not have explicit declarations at the beginning of a block; however, their types are well-defined. Labels are declared as such by the “;” which follows them [4.1.3]. Numbers
have type integer or real based on their syntax [2.5.4].
Expressions have a type which is dependent on the
constituent elements [3.3.4]. Hence, there is at least an
implicit declaration for each possible actual parameter; if
the declaration is incompatible with the specification, the
rule of case I applies; otherwise, the problem is solved by
application of case III.

Case III can be subdivided into case IIIa (in which the
formal parameter is called by value) and case IIIb (in
which the formal parameter is called by name).

In case IIIa the conflict can be resolved trivially by
noting [4.7.3.1] that a call by value has the effect of execut-
ing assignment statements and that transfer functions
between real and integer are defined in Algol 60 across
the ":=" in an assignment statement [4.2.4]. Hence, in
this case the conflict is only apparent and is resolved at
the time the procedure is invoked by invoking the appro-
priate transfer functions when the parameters called by
value are assigned to the local variables. Since [4.7.3.1]
these assignments take place before the body of the
procedure is entered, the parameters inside the procedure
body will have the type noted in the specification, while
the parameters outside the procedure body will have the type
noted in the declaration.

In case IIIb, the solution which seems to be most in-
keeping with the spirit of the original report is to consider
that if the specifications are optional for parameters called
by name [5.4.5] they may as well have not been written
at all; hence, if a conflict results in case IIIb (and not in
case I!) the specifications should be ignored and the
declaration obeyed.

As a specific example, consider the following:

\[
\begin{align*}
\text{begin} & \quad \text{integer } V; \\
\text{procedure } & \quad P(J); \text{ real } J; \\
\text{begin} & \quad \text{real } K; \\
& \quad J := 5.3; \\
& \quad V := V + 1; \\
& \quad K := J + 1; \\
\text{end } & \quad P; \\
& \quad P(V);
\end{align*}
\]

and ask the question "What is the value of K at the end
of the execution of the procedure?". According to the
solution proposed above, the answer is unambiguously
defined to be 7.

\[
\begin{align*}
\end{align*}
\]

In addition to the five major problems areas left un-
resolved at the Rome meeting, X3.4.2 examined other
aspects of Algol 60 (ROME) and suggests the following
resolutions:

a. **Function Designator Call Statement.** The
syntax of Algol permits a function designator to be used
as a procedure-call statement. This was inconsistent with
the rest of the language in that an expression was evaluated
and "left hanging." This is the only place in Algol that
such a thing occurs (an incomplete statement in some
meta-sense).

b. **Abnormal Exits from Function Designators.**
A function designator defines a value—either real, integer,
or Boolean—and hence it finds its place as a primary in
arithmetic and Boolean expressions. Algol permits,
however, a procedure body to be exited through a go to
nonlocal label. When this occurs, the function designator
may not have defined a value, and the expression of which
it formed a part is left incomplete. This seems definitely
undesirable. However, this concept has found wide
acceptance as an error-checking device, and probably
would find strong support if an attempt were made to
remove it from the language.

Therefore, we recommend the following position:
1. The value of a function designator is undefined if an
   exit is made from it other than through the final end.
2. If the value of a primary in an expression is un-
   defined, then the effect of evaluating that expres-
   sion is undefined.

c. **Numeric Labels.** Since the only additional facility
   introduced by numeric labels create an ambiguous case,
   numeric labels should be eliminated.

d. **Initialization of own Variables.** The own
   concept is incomplete without initialization of own
   variables and, therefore, such facility should be provided
   for explicitly in the language.

**RESEARCH SUMMARIES**

**Construction of Class-Teacher Time-Tables**
Institute of Computer Science, University of Toronto,
Toronto, Canada

Reported by: C. C. Gotlieb (Dec. 1962)

Descriptors: time-table, scheduling, assignment
problem, combinatorial analysis, Boolean matrices

The construction of class-teacher time-tables is a regular ac-
tivity in almost every university and large high school. In a
paper presented at IFIP-82 an algorithm for constructing such
time-tables is described. The method is based on an iteration over
sets of Boolean matrices which represent class and teacher availa-
bilities. A computer program for applying the algorithm has been
written and run successfully. This program is being used to in-
vestigate whether the algorithm yields all feasible time-tables,
as conjectured, and to determine how the convergence rate de-
pends on the size of problem.

A mailing list of about 50 persons or organizations interested
in this problem is available on request. Although published re-
sults are scarce, we are under way at many places. In several of
these, programs for assigning students to sections in a precon-
structed time-table are in regular use. The construction of time-
tables is related to problems in job shop assignments, network
flow and combinatorial set theory.